UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS ARARANGUÁ

Filipe Jerônimo Pereira

AVALIAÇÃO DA PLATAFORMA OPEN MPI PARA PARALELIZAÇÃO DO PROCESSO DE COMPILAÇÃO DE SOFTWARE

Araranguá, julho de 2014

Filipe Jerônimo Pereira

AVALIAÇÃO DA PLATAFORMA OPEN MPI PARA PARALELIZAÇÃO DO PROCESSO DE COMPILAÇÃO DE SOFTWARE

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como parte dos requisitos necessários para a obtenção do Grau de Bacharel em Tecnologias da Informação e Comunicação.

Orientador: Prof. Dr. Anderson Luiz Fernandes Perez

Araranguá, julho de 2014.

Filipe Jerônimo Pereira

AVALIAÇÃO DA PLATAFORMA OPEN MPI PARA PARALELIZAÇÃO DO PROCESSO DE COMPILAÇÃO DE SOFTWARE

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de Bacharel em Tecnologias da Informação e Comunicação, e aprovado em sua forma final pelo Curso de Graduação em Tecnologias da Informação e Comunicação.

Araranguá, julho de 2014.

Prof. Vilson Gruber, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Anderson Luiz Fernandes Perez, Dr.
Orientador

Prof. Gustavo Medeiros de Araújo, Dr.

Prof. Ricardo Alexandre Reinaldo de Moraes, Dr.

Este trabalho é dedicado aos meus amados pais Lucimar e Jiovane, e meus irmãos Rafael e Ricardo.

AGRADECIMENTOS

Agradeço aos meus pais e familiares que sempre me apoiaram e me incentivaram. Aos meus colegas de turma, dos quais dividimos muitos projetos durante nosso curso. Agradeço também ao meu professor e orientador Anderson Luiz Fernandes Perez, que me apoiou não somente neste trabalho, mas durante todo o curso nas disciplinas em que lecionou.

Agir, eis a inteligência verdadeira. Serei o que quiser. Mas tenho que querer o que for. O êxito está em ter êxito, e não em ter condições de êxito. Condições de palácio tem qualquer terra larga, mas onde estará o palácio se não o fizerem ali?

(Fernando Pessoa)

RESUMO

Os programas de computador estão cada vez maiores e mais complexos, exigindo maior capacidade de processamento do hardware. Uma alternativa para melhorar o tempo de execução de programas é o uso de sistemas multiprocessados com destaque à computação paralela. Em processos de desenvolvimento de software que usam o modelo em cascata o tempo de compilação pode se tornar um problema em projetos grandes, sobretudo ao atendimento dos prazos para a entrega do produto final, em virtude das várias etapas envolvidas neste processo. O uso de um ambiente de programação paralela pode tornar o tempo de compilação menor, uma vez que o processo de compilação será divididos em várias tarefas que serão executadas em processadores distintos. Neste trabalho são descritos os resultados obtidos na paralelização do processo de compilação de software utilizando a plataforma Open MPI que é uma implementação Open Source do padrão MPI (*Message Passing Interface*) e que fornece uma camada de abstração capaz de criar um ambiente computacional distribuído. Os resultados obtidos a partir da compilação de um sistema de teste na plataforma Open MPI são comparados com os resultados obtidos a partir da compilação do mesmo sistema em ambiente monoprocessado.

Palavras-chave: Open MPI, computação paralela, compilação.

ABSTRACT

Computer programs are becoming bigger and more complex requiring more processing hardware. An alternative to improve the execution time of programs is the use of multiprocessor systems with emphasis on parallel computing. Processes in software development using the waterfall model compilation time can become an issue in large projects, especially to deadlines for the delivery of the final product, because of the multiple steps involved in this process. The use of a parallel programming environment can reduce the compilation time, once the compilation process is divided into multiple tasks to be executed on different processors. In this work we describe the results obtained in the parallelization of software compilation process using Open MPI platform, an Open Source implementation of the MPI (Message Passing Interface), which provides an abstraction layer able to create a distributed computing environment. The results from the compilation of a test system in the Open Platform MPI are compared with the results obtained from the compilation of the same system in monoprocessor environment.

Keywords: Open MPI, parallel computing, compilation.

LISTA DE FIGURAS

Figura 1. Fases do processo de compilação de um programa de computador....34

Figura 2. O modelo SIMD. 41

Figura 3. O modelo MIMD. 42

Figura 4. Visão arquitetônica da camada de abstração do Open MPI mostrando suas três camadas principais: OPAL, ORTE e OMPI. 46

Figura 5. Visão arquitetônica do framework do Open MPI. 47

Figura 6. Estrutura de um código MPI em C. 49

Figura 7. Trecho do código fonte do programa de teste desenvolvido. 52

Figura 8. Listagem dos arquivos fonte da linguagem de programação Lua. 53

Figura 9. Mapeamento do diretório que contém os códigos fonte. 55

Figura 10. Tempo de execução de quatro configurações no ambiente de teste do primeiro experimento. 58

Figura 11. Média e desvio padrão dos tempos de execução dos programas de teste na primeira seção. 59

Figura 12. Tempo de execução de cinco configurações no ambiente de teste no segundo experimento. 61

Figura 13. Média e desvio padrão dos tempos de execução dos programas de teste no segundo experimento. 62

LISTA DE TABELAS

Tabela 1. Cronograma hipotético das etapas de desenvolvimento de software. 30

Tabela 2. *Tokens* que podem ser reconhecidos em C. 35

Tabela 3. Configuração dos computadores utilizados nos experimentos. 54

Tabela 4. Métricas de aceleração e eficiência. 56

Tabela 5. Cálculo do *Speedup* (aceleração) e eficiência na primeira seção. 60

Tabela 6. Cálculo do *Speedup* (aceleração) e eficiência. 62

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface* 45

BOINC *Berkeley Open Infrastructure for Network Computing* 43

IBM-PC *IBM Personal Computer* 42

IEEE *Institute of Electrical and Electronics Engineers* 43

MIMD *Multiple-Instruction Multiple-Data* 39

MISD *Multiple-Instruction Single-Data* 39

MPI *Message Passing Interface* 27

OMPI *Open MPI* 62

OPAL *Open Portable Access Layer* 62

ORTE *Open Run-Time Environment* 62

PFLOPS *Peta FLoating-point Operations Per Second* 58

SIMD *Single-Instruction Multiple-Data* 39

SISD *Single-Instruction Single-Data* 39

SUMÁRIO

1 INTRODUÇÃO 27

1.1 OBJETIVOS 29

1.1.1 Objetivo Geral 29

1.1.2 Objetivos Específicos 29

1.2 JUSTIFICATIVA E MOTIVAÇÃO 30

1.3 METODOLOGIA 31

1.4 ORGANIZAÇÃO DO TRABALHO 32

2 FUNDAMENTOS DO PROCESSO DE COMPILAÇÃO DE PROGRAMAS DE COMPUTADOR 33

2.1 DEFINIÇÃO DE COMPILAÇÃO 33

2.2 FASES DA COMPILAÇÃO 34

3 COMPUTAÇÃO PARALELA E DISTRIBUÍDA 39

3.1 SISTEMAS DISTRIBUÍDOS E PARALELOS 39

3.2 GRIDS E MULTICOMPUTADORES (CLUSTER) 42

3.3 OPEN MPI 44

3.3.1 Arquitetura da Camada de Abstração 46

3.3.2 Métodos da API 47

4 DESCRIÇÃO DO PROGRAMA DE TESTE 51

4.1 METODOLOGIA UTILIZADA PARA A AVALIAÇÃO DO PROCESSO DE COMPILAÇÃO COM O OPEN MPI 51

4.1.1 Descrição do Programa de Teste Desenvolvido 51

4.1.2 Escolha do Programa para Avaliação do MPI 52

4.1 AMBIENTE DE TESTE 54
4.2 DESCRIÇÃO DOS CRITÉRIOS DE AVALIAÇÃO 55
4.2.1 Obtenção do tempo de execução 56
5 RESULTADOS DA AVALIAÇÃO 57
5.1 APRESENTAÇÃO DAS CONFIGURAÇÕES DE TESTE 57
5.2 DESCRIÇÃO DOS EXPERIMENTOS 58
5.2.1 Experimento 1 58
5.2.2 Experimento 2 60
5.2.3 Considerações sobre os Experimentos Realizados 63
6 CONSIDERAÇÕES FINAIS 65
6.1 PROPOSTAS PARA TRABALHOS FUTUROS 66
REFERÊNCIAS 67
APÊNDICE A – Código fonte do programa de teste paralelo utilizando a API Open MPI 71

1 INTRODUÇÃO

A crescente demanda por processos gerenciados por programas de computador faz com que as suas funcionalidades cresçam exponencialmente. Desse modo os programas se tornam cada vez maiores e complexos, exigindo maior capacidade de processamento e manipulação da informação. Pereira (2006) destaca que o modo de produção de software passou de artesanal para um modo de produção racional, visando atender essa crescente demanda de softwares com maior qualidade e também à necessidade de se produzir em um curto intervalo de tempo.

Segundo as estatísticas da *Scientific American*, projetos de software ultrapassam em 50% o tempo estipulado no cronograma para a sua conclusão (HAZAN, 2004). O Standish Group (2013) possui levantamento estatístico sobre projetos de software baseado em uma base de dados com quase 50.000 projetos que provê uma visão global, tendo a maior concentração nos Estados Unidos e Europa, onde 60% dos projetos são dos Estados Unidos, 25% são da Europa e os 15% restantes são dos demais países. A pesquisa revela que 39% dos projetos são finalizados dentro do prazo e custos conforme previsto, 43% dos projetos são concluídos com atraso, orçamento superior e/ou menos recursos e funcionalidades do que o previsto, e 18% são cancelados.

Na construção de um programa, os algoritmos escritos em linguagens de programação como C, Java, Python, entre outras, necessitam de um processo de tradução da linguagem compreendida pelo homem, dita linguagem de alto nível, para a linguagem de máquina, a esse processo se dá o nome de compilação. Em um projeto de desenvolvimento de software o processo de compilação pode representar um problema para cumprir o prazo, pois se o modelo utilizado no projeto for do tipo cascata então a etapa de teste precisa aguardar que o processo de compilação seja concluído, e se esse atrasar então consequentemente todo o cronograma será prejudicado.

Diante do exposto acima e da limitação física e monetária para aumentar o desempenho dos processadores, distribuir uma tarefa serializada para mais de um computador pode ser uma solução viável para reduzir o tempo de conclusão da mesma sem grandes investimentos em infraestrutura. Como a maioria das corporações dispõem de uma rede interna de computadores, uma rotina que poderia ser otimizada pela técnica de paralelismo é a compilação, pois essa tarefa aplica processamento em uma série de arquivos para produzir uma única saída, podendo-se assim dividir o trabalho em cargas menores. Mesmo que haja dependência entre os arquivos, onde a ordem deve ser respeitada, um roteiro pode ser aplicado para que a rotina seja sincronizada, sendo necessário o conhecimento sobre a estrutura do programa a ser compilado.

O projeto Open MPI (*Message Passing Interface*) é uma implementação open source MPI-2, que é desenvolvido e mantido por um consórcio de acadêmicos, pesquisadores e parceiros da indústria. MPI é um conjunto de especificações de protocolos de comunicação de dados para computação paralela. O Open MPI é capaz de combinar o conhecimento, tecnologias e recursos de toda a comunidade de computação de alto desempenho, a fim de construir a melhor biblioteca MPI disponível, e oferece vantagens para fornecedores de sistemas e software, desenvolvedores de aplicativos e pesquisadores de ciência da computação (Open-Mpi.Org, 2014).

Com o Open MPI aplicado em um ambiente multiprocessado é possível criar uma infraestrutura capaz de executar rotinas de forma paralela. Com este recurso o problema do tempo elevado de compilação de software, pode ser reduzido ao distribuir a rotina em partes menores para ser executado em mais de um processador ao mesmo tempo.

1.1 OBJETIVOS

Para melhor compreensão deste trabalho os objetivos foram separados em objetivo geral e específicos.

1.1.1 Objetivo Geral

Avaliar a plataforma Open MPI para paralelização do processo de compilação de software.

1.1.2 Objetivos Específicos

1. Estudar os conceitos de computação paralela e distribuída;
2. Estudar o processo de compilação e arquitetura dos compiladores;
3. Selecionar e obter código fonte de uma aplicação *open source* de porte significativo para testes;
4. Estabelecer critérios de *benchmark* para testes de avaliação do processo de compilação baseado em Open MPI;
5. Avaliar os resultados obtidos a partir dos testes baseados nos critérios de *benchmark* estabelecidos.

1.2 JUSTIFICATIVA E MOTIVAÇÃO

Na indústria de software o tempo é um elemento muito significativo, pois todo o trabalho é mensurado em sua função, onde a correção de um erro, por exemplo, será convertido em “homens-horas”, e neste cálculo entram as etapas como levantamento de requisitos, desenvolvimento e testes. Um atraso no cronograma significa, como em qualquer tipo de indústria, perda não só de tempo, mas também monetária. A etapa de compilação é uma atividade recorrente e reduzir o seu tempo pode contribuir para o cumprimento do cronograma.

O tempo de compilação de um sistema simples geralmente leva alguns segundos, no máximo alguns minutos, mesmo que seja feito em um computador de uso doméstico. Porém, um sistema que envolve em seu código fonte diversas bibliotecas e milhares de linhas de código pode ter um tempo de compilação de algumas horas.

A Tabela 1 demonstra um cronograma em cascata considerando apenas o desenvolvimento e teste, onde são estimadas 5 (cinco) fases de teste até a liberação do produto final. Deste modo, pode-se verificar que se houver atraso na fase de desenvolvimento as demais etapas do cronograma serão afetadas, reduzindo a qualidade das próximas etapas, sendo necessário maior prazo.

Tabela 1. Cronograma hipotético das etapas de desenvolvimento de software.

Etapa	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Desenvolvimento																														
Teste																														
Desenvolvimento																														
Teste																														
Desenvolvimento																														
Teste																														
Desenvolvimento																														
Teste																														
Desenvolvimento																														
Teste/Liberação																														

O tempo de compilação representa um custo significativo para empresas de desenvolvimento de software que utilizam modelos de processos onde a etapa de testes precisa aguardar a finalização da etapa de compilação. Se ocorrer um erro durante a compilação, será necessário localizar e corrigir o código que causou a falha e então reiniciar o processo de compilação.

Contrariando a lei de Gordon Moore, onde a cada 18 meses a integração dos transistores dobraria, Coelho (2012, p. 25) afirma que atualmente o aumento da velocidade dos processadores apresenta limitações físicas, e também problemas como geração e dissipação de calor e alto consumo de energia. A indústria de hardware contorna esse problema aumentando o número de núcleos dentro de um mesmo chip, porém nem todas as aplicações estão prontas para fazer uso destes ambientes multiprocessados.

Resolver o problema do tempo elevado para o processo de compilação com tecnologias da computação distribuída não representa somente uma redução de custos, mas também uma atitude ecologicamente consciente, pois o melhor aproveitamento dos recursos computacionais representa menor consumo de energia e também menos lixo eletrônico gerado pela troca de equipamentos.

1.3 METODOLOGIA

A metodologia adotada neste trabalho será baseada em experimentos, onde será desenvolvido um programa de teste para executar de modo paralelo a compilação dos códigos fonte de uma aplicação *open source*, e sobre este programa de teste será mensurado os resultados obtidos utilizando os critérios de *benchmark* estabelecidos após estudo. Os testes envolverão ambientes multiprocessados comparando a diferença dos resultados ao executar o processo de compilação localmente e em rede.

Para realizar o desenvolvimento do programa de teste será realizado uma revisão bibliográfica para definir os conceitos de computação paralela e distribuída e de compiladores. Além destes conceitos, também será estudado a plataforma Open MPI.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em mais 5 (cinco) capítulos e um apêndice além da introdução.

O **Capítulo 2** apresenta a definição e descreve as etapas envolvidas no processo de compilação de um programa de computador.

O **Capítulo 3** contextualiza e define sistemas distribuídos e paralelos, apresenta o conceito de grids e multicomputadores (cluster) e também descreve a plataforma Open MPI.

O **Capítulo 4** descreve o programa desenvolvido para avaliação da compilação paralela, o ambiente de teste e os critérios a serem avaliados.

O **Capítulo 5** apresenta os resultados da avaliação obtidos a partir do programa de teste implementado com a plataforma Open MPI.

O **Capítulo 6** apresenta as considerações finais e propostas para trabalhos futuros.

O **Apêndice A** apresenta o código fonte do programa de teste desenvolvido com a API (*Application Programming Interface*) do Open MPI.

2 FUNDAMENTOS DO PROCESSO DE COMPILAÇÃO DE PROGRAMAS DE COMPUTADOR

Este capítulo apresenta a definição e descreve as etapas envolvidas no processo de compilação de um programa de computador.

2.1 DEFINIÇÃO DE COMPILAÇÃO

Da mesma forma que uma linguagem é utilizada para repassar à uma pessoa uma mensagem, os computadores também possuem uma linguagem própria que é utilizada para informar os passos que solucionarão um determinado problema. Esta "linguagem de máquina", conhecida como linguagem de baixo nível é representada por uma sequência de zeros e uns (AHO *et al.*, 1995).

Para facilitar a programação foram criadas linguagens de nível intermediário e alto nível através da definição de palavras reservadas e estruturas semânticas de compreensão mais amigável. Segundo Price e Toscani (2001) uma linguagem de alto nível representa o meio mais próximo ao problema a ser resolvido facilitando a escrita de programas.

Ao empregar uma linguagem de alto nível é necessário fazer a conversão para a linguagem de máquina, trabalho este feito pelo compilador, um programa ou um conjunto de programas que entende a entrada textual em forma de algoritmo (código fonte) e a transforma em outra linguagem semanticamente equivalente (código objeto). Esse processo então é chamado de compilação, pois são reunidos todos os elementos criados pelo programador para gerar uma saída que será passiva de execução por computador (PRICE e TOSCANI, 2001).

2.2 FASES DA COMPILAÇÃO

O processo de compilação é composto por duas etapas, análise e síntese. A etapa de análise é composta por análise léxica, sintática e semântica. A etapa de síntese é composta por gerador de código intermediário, otimizador de código e gerador de código objeto. A estrutura de um compilador é demonstrada na Figura 1, onde são destacadas as duas grandes etapas e o fluxo de operação entre elas.

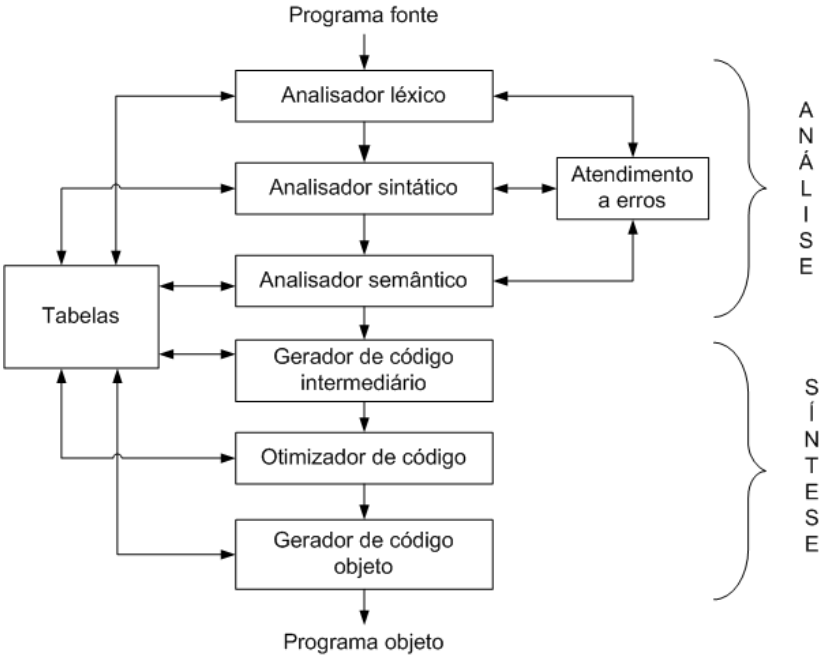

Figura 1. Fases do processo de compilação de um programa de computador.

Extraído de: (PRICE e TOSCANI, 2001)

A **análise léxica** ou linear faz a leitura do código fonte observando cada caracter identificando “tokens”, que são classes de símbolos como, identificadores, delimitadores e palavras reservadas. Caso seja identificado alguma unidade não aceita pela linguagem adotada então são lançadas mensagens de erro (AHO *et al.*, 1995).

A Tabela 2 lista os *tokens* que durante a análise léxica seriam encontrados em um programa de computador escrito em linguagem de programação C.

Tabela 2. *Tokens* que podem ser reconhecidos em C.

Palavras reservadas	if else while do
Identificadores	
Operadores relacionais	< > <= >= == !=
Operadores aritméticos	+ * / -
Operadores lógicos	&& || & | !
Operador de atribuição	=
Delimitadores	; ,
Caracteres especiais	( ) [ ] { }

Durante a leitura os caracteres não significativos como, espaços em branco e comentários, são desprezados, a menos que o espaço sirva para separar um identificador, constante ou palavra reservada adjacente (APPEL, 1998).

A **análise sintática** agrupa os *tokens* identificados na análise léxica em frases gramaticais, uma sequência de símbolos que representam uma estrutura sintática, como expressões e comandos (AHO *et al.*, 1995). O resultado é uma representação em árvore, conhecida como árvore de derivação, que obedece as regras estabelecidas pela linguagem utilizada (PRICE e TOSCANI, 2001).

O analisador sintático também é conhecido por *parser*. Para fazer a verificação utiliza uma gramática livre de contexto que compõe a

linguagem adotada e compara se os *tokens* repassados pela fase anterior formam uma representação aceitável.

Existem duas maneiras fundamentais de análise, a *top-down* e a *bottom-up*. Na maneira *top-down*, ou seja, de cima para baixo ou descendente, a árvore de derivação é construída da raiz a partir do símbolo inicial em direção as folhas. E na maneira *bottom-up*, ou seja, de baixo para cima ou crescente, a árvore é criada a partir dos *tokens* até chegar ao seu símbolo inicial (PRICE e TOSCANI, 2001).

Na **análise semântica** ocorre a verificação do significado das estruturas criadas no código fonte, segundo Price e Toscani (2001) o analisador sintático opera muitas vezes conjuntamente com o analisador semântico, pois depois de construir uma estrutura sintática é preciso verificar se a mesma realmente faz sentido. Por exemplo, na estrutura de seleção na linguagem C "IF <*expressao*> <*comando*>;" a expressão deve retornar um tipo lógico, ou seja, verdadeiro ou falso para que o comando a seguir seja ou não executado.

Segundo Appel (1998), na fase de análise semântica as definições de variáveis são conectadas ao seu uso, cada expressão é verificada se possui o tipo correto. A sintaxe abstrata gerada na etapa anterior é traduzida para uma representação adequada para gerar o código de máquina.

Para Price e Toscani (2001) além da avaliação dos operadores e operandos observando a correta aplicação, outra verificação importante é a de tipos, como por exemplo, se uma variável que representa números tem em algum momento atribuição de caracteres (*long := string*). Porém em algumas linguagens em determinadas situações a variável é convertida automaticamente, por exemplo, ainda na linguagem C se uma variável é declarada como do tipo inteiro e recebe um número real, este apenas perde as casas decimais na conversão sem apresentar erros na compilação ou execução.

As tabelas de símbolos são atualizadas nesta etapa, onde os identificadores são mapeados para seus tipos e locais. São processadas

as funções, declarações de tipos e variáveis, esses identificadores são atribuídos para "significados" na tabela de símbolos (APPEL, 1998).

A **geração de código intermediário** é a fase da compilação que produz, através da representação gerada pela etapa de análise, um código mais próximo do código objeto (baixo nível), onde as instruções de alto nível se multiplicam em algumas instruções de base, revelando todos os passos necessários para concluir operações que antes eram vistas como apenas uma palavra por exemplo.

Esta fase poderia ser omitida e ser gerado diretamente o código objeto, mas com ela existe algumas vantagens como a possibilidade de otimização do código enquanto ainda facilita o processo de tradução, que é mais complexo quando feito diretamente de alto nível para baixo nível (PRICE e TOSCANI, 2001).

A maior diferença entre o código intermediário e o código final é que no intermediário não são especificados os registradores, endereços de memória que serão utilizados ou outros itens equivalentes a estes. Existem várias formas de se representar o código intermediário, Aho *et al.* (1995) exemplificam o "código de três endereços", onde os endereços de memória representam os registradores. Neste modelo a sequência de instruções possuirá no máximo três operandos como no exemplo abaixo:

temp1 := intToReal (60)
*temp2 := id3 * temp1*
temp3 := id2 + temp2
id1 := temp3

O exemplo acima é resultado da seguinte representação obtida ao fim das etapas de análise: *id1 := id2 + id3 * 60.*

A **otimização de código** é a fase da compilação que tenta promover uma melhoria no código intermediário. Na otimização, linhas de instruções podem ser consideradas desnecessárias ao obter o mesmo resultado resumindo a operação em uma única linha. Ainda pode ser feito uma redução dos endereços de memória utilizados diminuindo a

movimentação entre registradores e recursos gastos no processamento (AHO *et al.*, 1995).

O exemplo anterior poderia ser escrito da seguinte maneira:

*temp1 := id3 * 60.0*
id1 := id2 + temp1

Desta forma foi utilizada apenas a localização de memória *temp1*, pois o compilador pode perceber que a conversão do número inteiro *60* pode ser feita diretamente em tempo de compilação já que o valor é uma constante, eliminando assim uma linha, além disso, foi utilizado *id1* para receber o resultado final ao invés de repassar para outra localização de memória primeiro, resultando em um fluxo de execução sintetizado (AHO *et al.*, 1995).

A **geração de código objeto** é a fase final da compilação, onde o código intermediário será traduzido para a linguagem de máquina, ou seja, de baixo nível. Nesta etapa as instruções são convertidas em comandos que efetuam operações compreendidas pelo processador, como movimentação entre registradores e cálculos aritméticos. A memória é reservada para variáveis e constantes e os registradores são selecionados (PRICE e TOSCANI, 2001). Segundo Aho *et al.* (1995) o código objeto consiste em código de máquina relocável ou código de montagem.

O código gerado deve realizar exatamente o proposto antes da tradução e ainda precisa utilizar os recursos disponíveis da melhor forma possível, consequentemente, essa é uma das fases mais difíceis das etapas de compilação.

3 COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Este capítulo contextualiza e define sistemas distribuídos e paralelos, apresenta o conceito de grids e multicomputadores (cluster) e também descreve a plataforma Open MPI.

3.1 SISTEMAS DISTRIBUÍDOS E PARALELOS

Por volta dos anos 70 quando os primeiros computadores começaram a ser interligados já se pensava na ideia de compartilhar os recursos computacionais ociosos. Porém nos anos 90 as pesquisas se intensificaram buscando solucionar problemas onde uma única máquina era insuficiente para o processamento. A partir de softwares desenvolvidos nesta época se tornou possível utilizar até milhares de máquinas distribuídas geograficamente através da internet para um mesmo propósito (DANTAS, 2005).

Tanenbaum e Steen (2002) definem sistemas distribuídos como uma coleção de computadores independentes que aparentam para o usuário como um sistema único e coerente. Essa definição conota a necessidade de autonomia dos componentes (computadores) do sistema, além da necessidade de colaboração para a formação de um único sistema. Segundo Coulouris (2007) um sistema distribuído possui seus componentes em computadores conectados em rede e agem coordenadamente através de mensagens. As principais características de um sistema distribuído são: ausência de um relógio global para sincronização, concorrência de componentes e falhas de componentes independentes.

Conforme constata Dantas (2005) os grandes avanços no desenvolvimento dos processadores e memórias que vinha ocorrendo nas ultimas décadas, demonstra queda devido à restrições da própria

física. Este fato motiva projetos sobre computação distribuída, sendo uma tentativa diferenciada e interessante sobre vários aspectos.

Dantas (2005) discute duas configurações de infraestrutura computacionais, *Clusters* e *Grids*, observando um contexto denominado como computação distribuída de alto desempenho, que tem por objetivo melhorar o desempenho de aplicações distribuídas e paralelas.

Aplicações distribuídas se beneficiam dos recursos compartilhados que estão ociosos a fim de melhorar seu desempenho, porém estes recursos não são necessariamente relacionados. Diversas aplicações podem ser executadas neste contexto onde algumas apresentaram melhor desempenho, entretanto outras não, justamente por não haver relação entre elas.

Aplicações paralelas por outro lado, constituem uma divisão da tarefa que será executada em partes menores e distribuída em uma infra-estrutura computacional para execução lado a lado, onde o resultado depende da soma de todas as partes ao final. Diferente das aplicações distribuídas é indispensável que todas as partes sejam processadas com sincronismo e objetivando maior desempenho (DANTAS, 2005).

Segundo Michael Flynn os sistemas de computação paralela e distribuída podem ser classificados conforme número de fluxo de instruções e número de fluxo de dados. Esta classificação conhecida por taxonomia de Flynn foi concebida em 1966 e compreende quatro tipos. Quando há apenas um único fluxo de instruções e um único fluxo de dados, o sistema é classificado como SISD (*Single-Instruction Single-Data*). O sistema composto por apenas um fluxo de instruções e múltiplos fluxos de dados é classificado como SIMD (*Single-Instruction Multiple-Data*) e o contrário, como MISD (*Multiple-Instruction Single-Data*). A arquitetura mais abrangente é o sistema MIMD (*Multiple-Instruction Multiple-Data*) (CÁCERES *et al.*, 2001).

Na prática essa classificação se refere ao número de processadores empregados e se eles são autônomos ou se recebem a instrução de uma única unidade de controle, e ainda se possuem memória individual ou compartilhada. O MPI é adequado para

aplicações desenvolvidas para os sistemas classificados como SIMD e MIMD (COELHO, 2012).

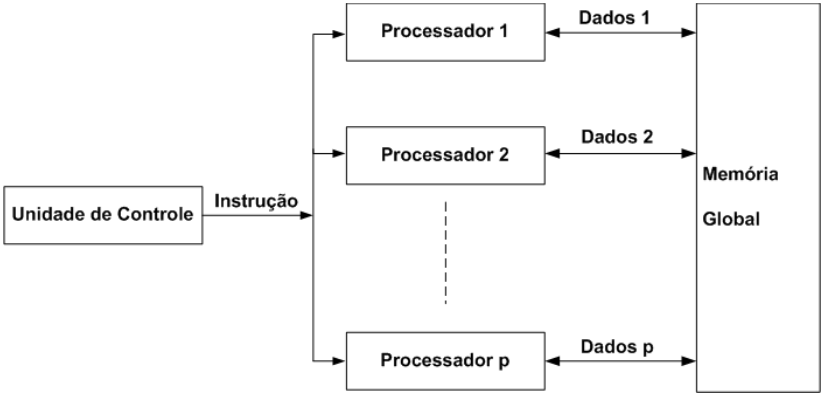

Figura 2. O modelo SIMD.
Extraído de: (Cáceres *et al.*, 2001)

A Figura 2 representa a arquitetura SIMD, onde os processadores recebem as instruções de uma única unidade de controle, executando assim em paralelo a mesma aplicação, podendo manipular dados diferentes através de uma memória compartilhada. A instrução recebida pelos processadores é a mesma, porém os dados utilizados nas operações podem ser diferentes (CÁCERES *et al.*, 2001).

A Figura 3 representa a arquitetura MIMD, onde os processadores são independentes, contendo cada um sua própria unidade de controle e memória. As operações são executadas interligadas através de uma rede e podem ser do tipo síncrona ou assíncrona. Os elementos trabalham para um mesmo objetivo, porém podem executar instruções diferentes sobre dados diversos (CÁCERES *et al.*, 2001).

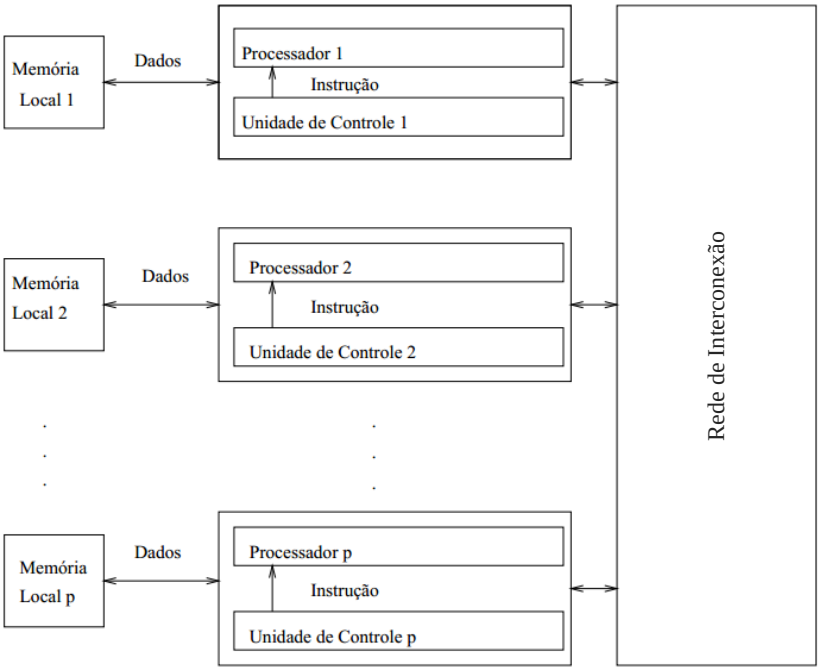

Figura 3. O modelo MIMD.
Extraído de: (CÁCERES *et al.*, 2001)

3.2 GRIDS E MULTICOMPUTADORES (CLUSTER)

A configuração de infraestrutura computacional para aplicações distribuídas chamada de *Cluster* compreende um ambiente que opera dentro de uma pequena área geográfica com o agrupamento físico ou virtual de vários computadores. Apesar de existir equipamentos especiais para esse tipo de infraestrutura, computadores do tipo IBM-PC também podem formar um *Cluster*. Para conexão virtual um pacote de software fará o intermédio provendo que os recursos computacionais de determinado computador fique passivo de receber solicitações de terceiros (Dantas, 2005).

Coelho (2012) acrescenta na definição de *Cluster* ou aglomerado de computadores, o qual o usuário final verá como uma única "imagem", corroborando com a definição de Tanenbaum e Steen (2002) sobre sistemas distribuídos. Cada computador nesta configuração é chamado de nó, onde um destes será responsável por controlar e gerenciar os demais, recebendo o nome de mestre. Alguns itens são importantes para garantir a confiabilidade deste tipo de sistema, como por exemplo, tolerância a falhas (caso um nó falhe os demais continuam o trabalho sem interrupção), escalabilidade (possibilidade de fazer o crescimento da estrutura de forma eficiente) e disponibilidade (tempo que o sistema permanece pronto para receber requisições).

Além da finalidade abordada por Dantas (2005), Coelho (2012) descreve mais duas:

- Alta disponibilidade: são configurações com aplicações capazes de identificar falhas no sistema e prover um tempo de religação (*uptime*) eficientemente e de forma automática.
- Balanceamento de carga: são configurações que monitoram o esforço das unidades que formam o sistema e gerenciam as requisições para evitar sobrecarga de um único nó, fazendo o trabalho de escalonamento dos processos.

A configuração de infraestrutura computacional para aplicações distribuídas chamada de *Grid* compreende um ambiente que opera dentro de uma grande área geográfica através de vários computadores interligados de forma a compartilhar seus recursos. A diferença entre *Grid* e *Cluster* é que a *Grid* não se restringe a uma organização, pois sua abrangência pode ser dentro de uma cidade, país ou até mesmo continentes. A formação de um *Grid* proporciona de forma cooperativa uma estrutura que fornece recursos e serviços como, processamento, armazenagem, acesso a dispositivos especiais, etc. (DANTAS, 2005).

Coelho (2012) aproxima a definição de *Grid* com a definição de aplicação paralela, dizendo que uma *Grid* compreende computadores de domínios diferentes que tem por objetivo melhorar o desempenho de uma tarefa dividida em partes menores entre os nós que compõem a

configuração fazendo ao final a junção destas partes para formar o resultado. Este tipo de infraestrutura utiliza um *middleware* para realizar este trabalho. Um dos mais bem sucedidos projetos *open source* para computação em *Grid* é o *BOINC* (*Berkeley Open Infrastructure for Network Computing*), utilizado para analisar ondas de rádio a fim de descobrir comunicação extraterrestre, sendo que os elementos que formam o agrupamento são de voluntários que doam os recursos computacionais ociosos para este projeto (BOINC, 2014).

3.3 OPEN MPI

Open MPI é uma implementação da especificação MPI (*Message Passing Interface*). O protocolo é uma unificação dos trabalhos realizados pelos integrantes do MPI fórum (http://www.mpi-forum.org) sobre computação paralela e distribuída ainda na década de 90 (GROPP, 1996).

O MPI não é uma norma homologada por institutos renomados como, por exemplo, IEEE (*Institute of Electrical and Electronics Engineers*), mas mesmo assim consiste em um padrão com propriedade para arquiteturas de memória distribuída, promovendo rotinas de distribuição e consolidação dos dados, sincronização e controle de processos entre os computadores que compõem uma infraestrutura distribuída (SNIR *et al.*, 1998).

A primeira versão da especificação contendo a base foi publicada em 1994, denominada de MPI-1, sendo melhorada em 1997 com as questões mais complexas na versão MPI-2. Somente em 2008 e 2009 novas atualizações foram realizadas, corrigindo erros sendo então lançadas as publicações MPI-2.1 e MPI-2.2. O trabalho mais recente publicado pelo grupo foi nomeado de MPI-3.0 e consiste em uma das maiores atualizações realizadas no padrão (COELHO, 2012).

Cáceres *et al.* (2001) lista as características do padrão MPI que o torna tão aceito e promissor para soluções computacionais distribuídas:

- Implementações gratuitas e de boa qualidade;
- Comunicação assíncrona;
- Gerenciamento eficiente de *buffers* de mensagens;
- Alta portabilidade;
- Especificação formal.

Uma implementação da especificação MPI oferece para o programador uma camada de abstração de alto nível que faz a intercomunicação entre as tecnologias de rede gerenciadas pelo sistema operacional e a aplicação. O objetivo principal é fornecer uma interface simples para desenvolver aplicações com alto desempenho principalmente nas linguagens Fortran, C e C++, sendo altamente portável. Existem implementações livres como o Open MPI e MPICH, mas também há implementações proprietárias como Intel MPI, MATLab MPI e HP MPI (COELHO, 2012).

Para se obter aumento de desempenho em uma aplicação originalmente serial, devem-se identificar os pontos no código que podem ser executados simultaneamente por outros processos, sendo que estes trechos devem ser envolvidos por um bloco de instruções disponibilizado pelo padrão MPI (COELHO, 2012). Além disso, diminuir o número de troca de mensagens necessárias para concluir a rotina, também contribuirá para obtenção de maior desempenho, já que o tempo gasto para distribuição e consolidação das informações será minimizado (MORALES, 2008).

3.3.1 Arquitetura da Camada de Abstração

Open MPI possui três camadas de abstração conforme destacado na Figura 4.

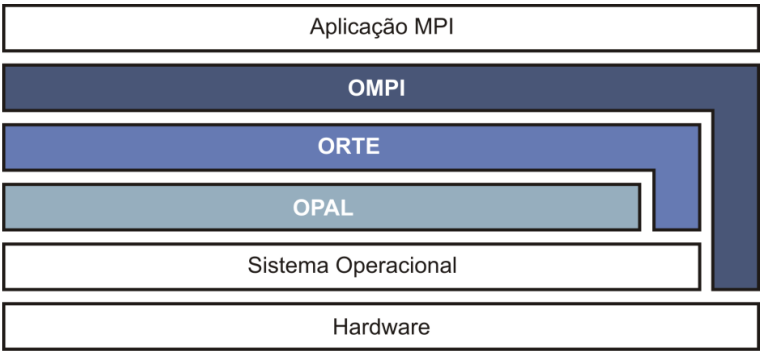

Figura 4. Visão arquitetônica da camada de abstração do Open MPI mostrando suas três camadas principais: OPAL, ORTE e OMPI. Adaptado de: (Squyres, 2014)

A camada *Open Portable Access Layer* (OPAL) provê funcionalidades como manipulação de string, controles de depuração entre outras. Também é nesta camada que está o núcleo para portabilidade entre diferentes sistemas operacionais, contendo interfaces de descoberta de IP, cronômetros de alta precisão, compartilhamento de memória entre processos do mesmo servidor, etc.

Open Run-Time Environment (ORTE) oferece um sistema de tempo real para iniciar, monitorar, e finalizar tarefas paralelas. ORTE usa *rsh* ou *ssh* para iniciar processos individuais em tarefas paralelas.

Open MPI (OMPI) é a camada de abstração mais alta, única com acesso direto por aplicações. Nesta camada está implementado o que o padrão MPI define. Suporta uma vasta variedade de tipos de rede e protocolos para oferecer portabilidade (Squyres, 2014).

A Figura 5 ilustra alguns frameworks e components do Open MPI. O exemplo contém frameworks das três camadas, OMPI (*btl* e *coll*), ORTE (*plm*) e OPAL (*timer*). Cada framework tem além da *base* um ou mais componentes.

A maior vantagem na abordagem de *plugins* é a liberdade para os desenvolvedores utilizarem diferentes implementações sem precisar alterar o núcleo do Open MPI (Squyres, 2014).

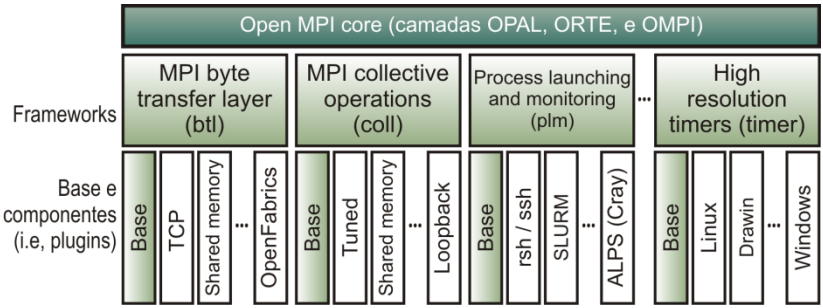

Figura 5. Visão arquitetônica do framework do Open MPI.
Adaptado de: (Squyres, 2014)

3.3.2 Métodos da API

No padrão MPI o endereçamento de memória entre os processos não é compartilhado, ou seja, uma variável está visível apenas para o processo que fez a alocação, e desta forma para alterar ou obter dados entre os diversos fluxos de execução que serão criados são utilizados métodos de envio e recebimento (Snir *et al.*, 1998). Este modelo de comunicação recebe o nome de comunicação ponto a ponto.

As bibliotecas do Open MPI possuem diversas funções, porém para a construção de um programa paralelo simples apenas seis são o suficiente para as rotinas fundamentais (Morales, 2008).

Alguns conceitos importantes para entendimento das funções presentes no MPI e para entender o funcionamento do fluxo gerado ao utilizá-lo, são listados por Coelho (2012):

- **Processo:** o programa como um todo que poderá ser dividido em mais de uma parte para ser executado em mais de um processador.
- **Grupo:** um conjunto de processos. A constante MPI_COMM_WORLD mantém o comunicador dos processos definidos na aplicação distribuída pelo padrão MPI.
- ***Rank:*** identificação do processador para controle da execução do programa, consistindo de um número inteiro, definido pela função MPI_Comm_rank para cada processo em um comunicador.
- ***Comunicador:*** componente em uma aplicação que representa o domínio de uma comunicação.
- ***Buffer* de Aplicação:** espaço reservado de memória, gerenciado pela aplicação, usado para armazenar os dados das mensagens que um processo precisa enviar ou receber.
- ***Buffer* de Sistema:** espaço reservado de memória pelo sistema para gerenciar as mensagens.

3.3.2.1 Inicialização e finalização MPI

Para utilizar os métodos e as constantes que compõem a API é preciso incluir a biblioteca *mpi.h*. Todos os métodos devem estar entre a área paralela do código, definida pelos métodos *MPI_Init* e *MPI_Finalize* (Coelho, 2012). O método *MPI_Init* possui dois parâmetros, sendo nesta ordem, um inteiro com o número de argumentos e um vetor de caracteres com os argumentos (Open-Mpi.Org, 2014). Os argumentos utilizados são os mesmos informados

para a chamada *main* em C. Abaixo é apresentado um modelo básico do código em C destacando o descrito.

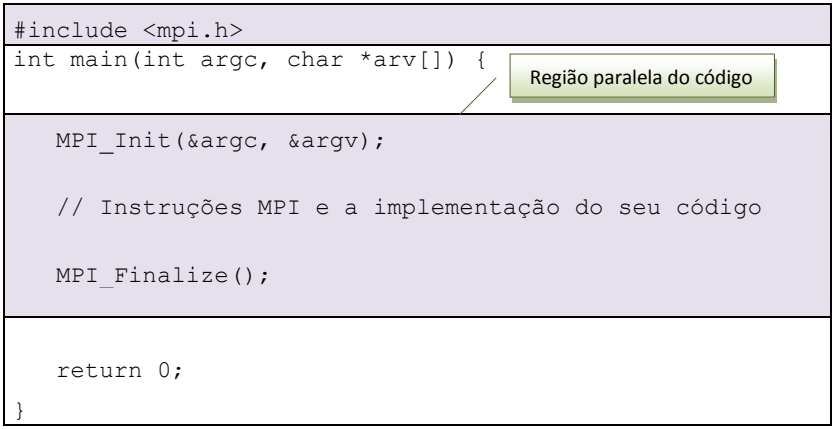

Figura 6. Estrutura de um código MPI em C.
Extraído de: (Coelho, 2012)

A chamada *MPI_Init* inicializa o ambiente de execução utilizando os parâmetros informados ao usar a ferramenta *mpirun* que está presente no pacote do Open MPI, assim como as ferramentas de compilação *mpicc* e *mpic++* para códigos em C e C++. A chamada de inicialização deve ser feita uma única vez até que seja concluída pelo método *MPI_finalize*, do contrário será retornado o código de erro *MPI_ERR_OTHER*. O parâmetro que irá definir o número de processos é *-np* e um inteiro com o valor desejado (Snir *et al.*, 1998). O exemplo abaixo ilustra a sequência de instruções para a execução do MPI.

mpirun –np 4 ./programa

3.3.2.2 Mensagens síncronas

Os métodos para comunicação ponto a ponto entre os processos são *MPI_Send* e *MPI_Recv*, para envio e recebimento nesta ordem. Estes métodos são síncronos porque a próxima instrução do código só será executada após a confirmação do recebimento da mesma (Coelho, 2012). A comunicação efetuada por estes médotos pode ser definida apenas como uma transferência de um conjunto de bytes de um processo para outro (Morales, 2008).

A assinatura do método *MPI_Send* é:

int MPI_Send(void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

Os parâmetros: ***buf***, endereço de memória dos dados que serão transmitidos, ***count***, número de elementos da mensagem, ***datatype***, especificação do tipo de dados, ***dest***, identificador do processo que irá receber a mensagem ou a constante *MPI_ANY_SOURCE* indicando que qualquer destino pode receber a mensagem, ***tag***, um rótulo que pode ser usado como filtro ao receber a mensagem aceitando apenas uma *tag* específica, ***comm***, comunicador que pode ser a constante *MPI_COMM_WORLD* em um sistema SIMD ou o comunicador criado pelo método *MPI_Comm_create* (Morales, 2008).

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

O método *MPI_Recv* tem apenas um argumento a mais, ***status***, este possui informações da mensagem recebida como a origem.

4 DESCRIÇÃO DO PROGRAMA DE TESTE

Este capítulo descreve o programa desenvolvido para avaliação da compilação paralela, o ambiente de teste e os critérios a serem avaliados.

4.1 METODOLOGIA UTILIZADA PARA A AVALIAÇÃO DO PROCESSO DE COMPILAÇÃO COM O OPEN MPI

A metodologia para a avaliação do processo de compilação com a plataforma Open MPI consiste em realizar alguns experimentos com o programa de teste desenvolvido com a API do Open MPI, onde o mesmo recebe a entrada de arquivos contendo código fonte e então realiza o processo de compilação.

4.1.1 Descrição do Programa de Teste Desenvolvido

Para avaliar a plataforma Open MPI para paralelizar a compilação de software foi desenvolvido um programa em linguagem de programação C que obtém a lista de arquivos do programa a ser compilado, e então faz a divisão da lista através da razão do número total de arquivos pelo número total de processos. Cada processo terá definido um intervalo da lista e fará a chamada do compilador GCC (*GNU Compiler Collection*) para cada um dos arquivos, ou seja, fará a tradução do código fonte em código objeto. A tarefa será distribuída em *n* processos e *m* hosts de acordo com os parâmetros informados ao executar o programa.

O programa de teste foi desenvolvido com o uso da biblioteca *mpi.h* fornecida pelo pacote do Open MPI. A Figura 7 apresenta um trecho do código do programa desenvolvido. O código completo está listado no Apêndice A.

```
int main(int argc, char *argv[])
{
    /*(...)*/
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &tid);
    MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
    /*(...)*/
    lstrListaArquivos = get_lista_arquivos_c();
    liControle  = lstrListaArquivos.liContador / nthreads;
    liResto     = lstrListaArquivos.liContador % nthreads;
    /*(...)*/
    liFim       = (tid + 1) * liControle;
    liInicio    = liFim - liControle + 1;
    if((tid + 1 == nthreads) && liResto > 0) liFim += liResto;
    /*(...)*/
    for(liIndice = liInicio; liIndice <= liFim; liIndice ++){
        /*(...)*/
        strcpy(lsComando, "gcc -c ");
        strcat(lsComando, lstrListaArquivos.lsArquivos[liIndice]);
        llRetorno = system(lsComando);
        /*(...)*/
    }
    MPI_Finalize();
    return(0);
}
```

Figura 7. Trecho do código fonte do programa de teste desenvolvido.

4.1.2 Escolha do Programa para Avaliação do MPI

Por questões legais e de integridade intelectual os testes não puderam ser realizados sobre o código fonte da aplicação comercial em que o autor contribui no setor de desenvolvimento em empresa privada.

Portanto, foi selecionada uma aplicação com código fonte sob a licença de software livre. Também foi observada a complexidade de ligação com as bibliotecas do código fonte, pois os arquivos precisam estar no mesmo diretório para não necessitar especificação explícita nos argumentos repassados para o compilador.

Os códigos fontes utilizados para os testes são da implementação da linguagem de programação Lua. Um projeto desenvolvido na linguagem C com cerca de 20.000 linhas de código na versão 5.2.3. A Figura 8 ilustra o conjunto de arquivos fonte que fazem parte do projeto Lua.

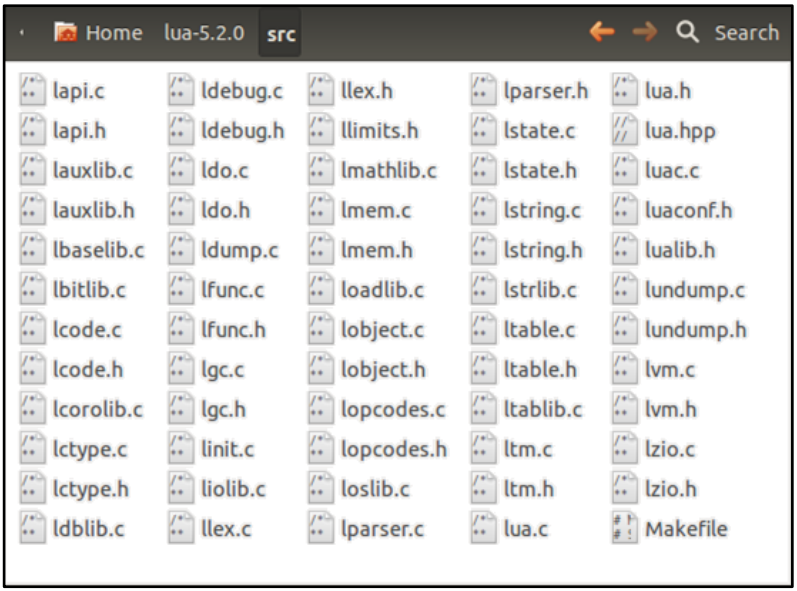

Figura 8. Listagem dos arquivos fonte da linguagem de programação Lua.

O projeto Lua foi desenvolvido pelo Tecgraf, Grupo de Tecnologia em Computação Gráfica da PUC-Rio e atualmente possui laboratório próprio denominado LabLua no Departamento de Informática da PUC-Rio.

Para intensificar os testes os arquivos da implementação da linguagem de programação Lua foram duplicados, dobrando assim o número de linhas de código para a primeira experiência e após duplicado novamente para a segunda experiência, obtendo-se o quádruplo do número de arquivos.

4.1 AMBIENTE DE TESTE

O ambiente de teste foi composto de dois computadores interligados por uma rede ponto a ponto, com a velocidade de conexão de 1 Gb/s através de cabeamento do tipo par trançado conectado diretamente computador à computador. Ambos com o sistema operacional Linux instalado distribuição Ubuntu 12.04 LTS. A rede foi configurada utilizando recursos dos pacotes *openssh* e *samba*. A Tabela 3 lista a configuração detalhada de cada computador utilizado.

Tabela 3. Configuração dos computadores utilizados nos experimentos.

Descrição	**Notebook I3**
Processador	Intel I3 – 2.10 GHz
Memória RAM	2 GB
Adaptador de rede	Ethernet Gigabit 10/100/1000
Descrição	**Desktop Core2Duo**
Processador	Intel Core2Duo – 2.60 GHz
Memória RAM	2 GB
Adaptador de rede	Ethernet Gigabit 10/100/1000

Para minimizar o tráfego na rede e simplificar a implementação foi estabelecido um mapeamento do diretório com o programa e códigos fonte presente no **desktop** tornando o acesso à mesma para o **notebook** como se fosse um diretório local, conforme Figura 9.

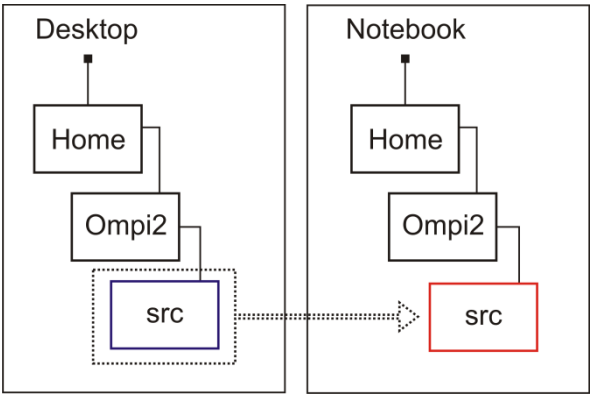

Figura 9. Mapeamento do diretório que contém os códigos fonte.

Este mapeamento permite que o programa de teste ao ser executado também no *Notebook* faça a leitura dos arquivos com os códigos fonte e escrita dos arquivos com código objeto após a compilação diretamente no *Desktop.*

4.2 DESCRIÇÃO DOS CRITÉRIOS DE AVALIAÇÃO

Segundo Queiroz (2007) para verificar o desempenho de uma aplicação paralela é necessário definir as métricas de desempenho sob alguns aspectos como: objetivo da avaliação, as características que envolvem o objeto de estudo e as características da carga de trabalho gerada.

Coelho (2012) explica que ao vislumbrar um modelo paralelo de computação há uma tendência em concluir que o aumento de desempenho será exponencial em fator do número de processadores empregados, porém na prática isto nem sempre é possível, pois existe uma série de aspectos que influenciam no resultado final. Como exemplo desses aspectos pode-se citar o tempo de inicialização do

ambiente paralelo, conexão entre pontos de rede, escalonamento de processos, utilização assertiva do buffer de comandos recentes e mais utilizados, entre outros.

Neste trabalho a métrica para definição do desempenho foi baseada no tempo de execução total da versão paralela do programa em comparação com a versão serial. Para a comparação foi adotado a média de 10 (dez) baterias de testes em ambos os ambientes.

Para comprovar o ganho em desempenho obtido pelo programa na versão paralela foi utilizado a métrica de *Speedup* ou Aceleração (Sp), definida pela divisão do tempo total do programa serial (Ts) pelo tempo total gasto pelo programa paralelo (Tp), o que resulta na fórmula Sp = Ts/Tp (BACELLAR, 2010).

Com a aceleração calculada pode-se verificar a eficiência (Ef), métrica definida através da razão entre a aceleração (Sp) e o número de processadores (p), resultando na fórmula Ef = Sp/p (BACELLAR, 2010). A avaliação dos resultados é verificada com os parâmetros da Tabela 4.

Tabela 4. Métricas de aceleração e eficiência.

Caso	Aceleração (*Sp*)	Eficiência (*Ef*)
Ideal	= p	= 1
Real	< p	< 1
Excepcional	> p	> 1

4.2.1 Obtenção do tempo de execução

Os tempos de execução do programa de teste foram obtidos através do comando *time*, um comando presente nos sistemas operacionais baseados no Unix. Ao incluir o comando *time* antecedendo os comandos de execução do programa de teste, ao final é apresentado o tempo total utilizado para o processamento da rotina de compilação.

5 RESULTADOS DA AVALIAÇÃO

Este capítulo apresenta os resultados da avaliação obtidos a partir do programa de teste implementado com a plataforma Open MPI.

5.1 APRESENTAÇÃO DAS CONFIGURAÇÕES DE TESTE

Foram definidas 5 (cinco) configurações para comparar o comportamento da implementação do Open MPI sobre o ambiente de testes em diferentes situações: MPI 2p, MPI 4p, MPI-Local 2p, MPI-Local 4p e Local.

A configuração "**MPI 2p**" se refere a versão paralela do programa de teste tendo como argumento o endereço de rede de cada nodo descrito no capítulo 4, e "**MPI 4p**" se refere a mesma situação porém com a adição do argumento de execução *–np 4* que inicializa quatro processos. A configuração "**MPI-Local 2p**" se refere a execução local do programa de teste na versão paralela, ou seja, foi executado sem a definição do endereço dos nodos mas com o argumento que inicializa dois processos, assim como a configuração "**MPI-Local 4p**", porém, inicializando 4 processos. E por fim a configuração "**Local**" se refere a execução da versão local do programa de teste que não utiliza a API do Open MPI.

Foram realizados dois tipos de experimentos, sendo que o primeiro utiliza o dobro dos arquivos do programa a ser compilado, e o segundo utiliza o quádruplo dos arquivos, que foram obtidos através da cópia dos originais.

5.2 DESCRIÇÃO DOS EXPERIMENTOS

Esta seção apresenta os resultados obtidos com os experimentos realizados com a plataforma Open MPI. A Seção 5.2.1 descreve o primeiro experimento realizado e a Seção 5.2.2 o segundo experimento.

5.2.1 Experimento 1

Neste experimento os arquivos com o código fonte do projeto Lua foram duplicados e submetidos ao programa de teste em diferentes configurações multiprocessadas para comparação com a configuração monoprocessada.

A Figura 10 apresenta a diferença dos tempos obtidos em 10 (dez) baterias de testes em quatro configurações.

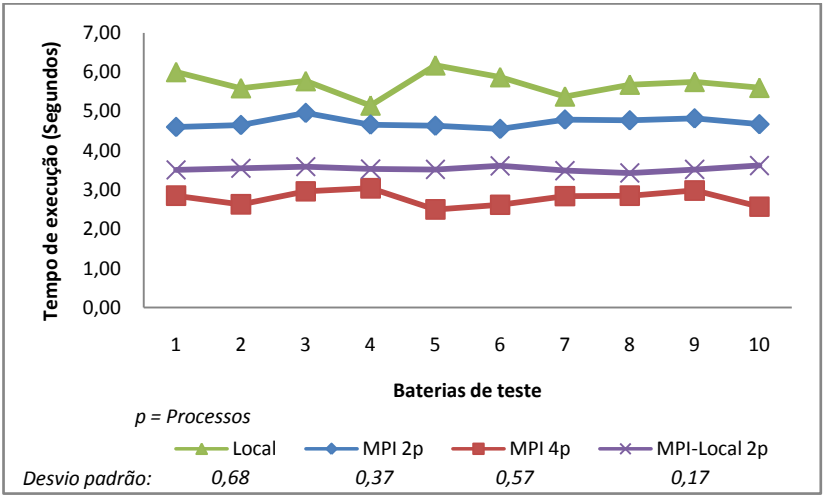

Figura 10. Tempo de execução de quatro configurações no ambiente de teste do primeiro experimento.

Ao executar o programa de teste na versão paralela com dois processos (MPI 2p), dividindo a metade dos arquivos que foram compilados para cada um dos computadores descritos no Capítulo 4, observa-se que todos os resultados foram abaixo da versão local. A configuração paralela executada localmente com dois processos (MPI-Local 2p) teve tempo abaixo da configuração executada em dois nodos (MPI 2p), o que demonstra o atraso gerado pela comunicação e gerenciamento dos processos em mais de um computador. O melhor resultado foi obtido pela configuração com quatro processos (MPI 4p), utilizando todos os processadores físicos disponíveis no ambiente de teste.

A Figura 11 apresenta a média e o desvio padrão dos tempos obtidos com as quatro configurações.

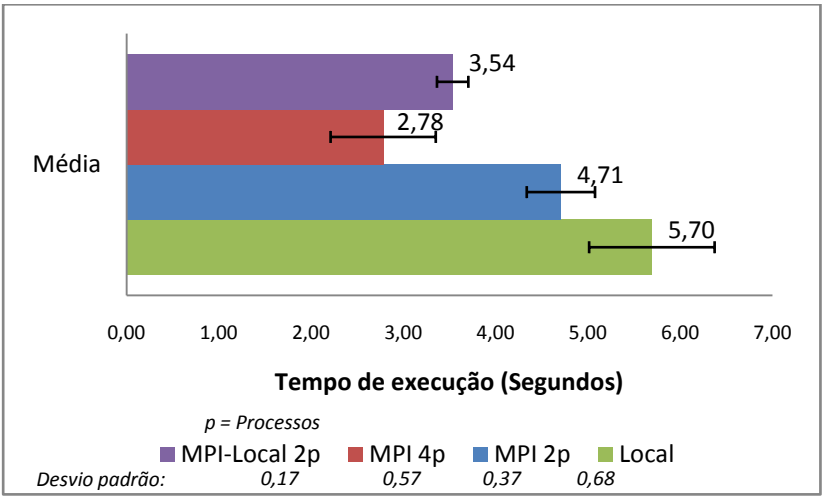

Figura 11. Média e desvio padrão dos tempos de execução dos programas de teste na primeira seção.

A média da configuração *MPI 2p* apresenta menos de 1 (um) segundo abaixo da configuração *Local.* A configuração *MPI 4p* obteve o

melhor resultado, utilizando menos da metade do tempo necessário pela configuração *Local.*

Tabela 5. Cálculo do *Speedup* (aceleração) e eficiência na primeira seção.

Configurações	*Média*	*D. Padrão*	*Speedup*	*Eficiência*
MPI 2p	5,07	0,37	1,21x	0,61
MPI 4p	2,78	0,57	2,05x	0,51
MPI-Local 2p	3,54	0,17	1,61x	0,81
Local = 5,70			*Sp = Ts/Tp*	*Ef = Sp/p*

Na Tabela 5 é possível observar que a configuração *MPI 4p* com a melhor aceleração também tem a menor eficiência. Isso ocorre porque o número de processos empregados para a resolução do problema não resulta na redução do tempo no mesmo fator. Por outro lado, a configuração *MPI-Local 2p* teve maior eficiência, ficando muito próximo ao ideal.

A configuração *MPI 2p* não demonstrou ser interessante no ambiente de teste, com baixa aceleração e baixa eficiência, sendo melhor neste caso utilizar a configuração paralela localmente.

5.2.2 Experimento 2

Neste experimento foi incluído a configuração *MPI-Local 4p* para testar a capacidade dos processadores virtuais disponíveis pela tecnologia *hyperthreading*.

A Figura 12 ilustra os tempos de execução das 5 (cinco) configurações em 10 (dez) baterias de teste.

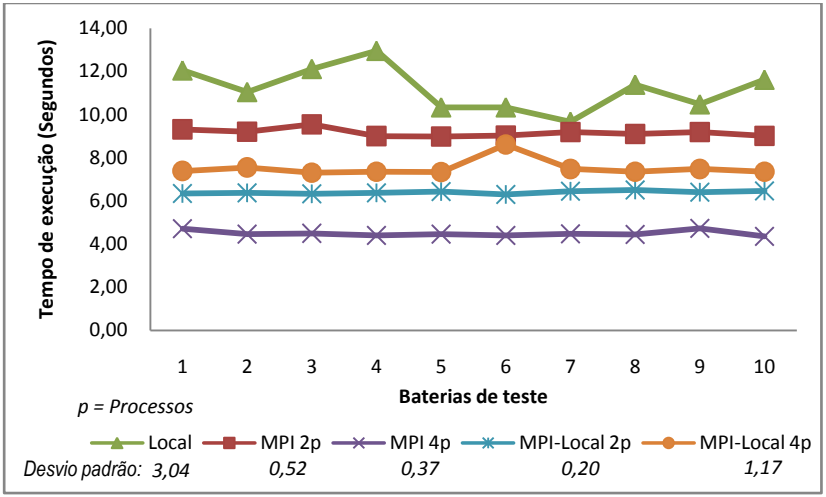

Figura 12. Tempo de execução de cinco configurações no ambiente de teste no segundo experimento.

Neste experimento os tempos foram semelhantes ao anterior. As configurações *MPI 4p* e *MPI-Local 2p* são as únicas que apresentam linearidade nos resultados, indicando melhor ajuste no ambiente de teste.

A configuração *MPI-Local 4p* apresentou resultados piores que a configuração *MPI-Local 2p*, o que demonstra que para este problema é mais custoso gerenciar 4 processos, onde a divisão da tarefa considerando os processadores virtuais não é interessante neste ambiente de teste.

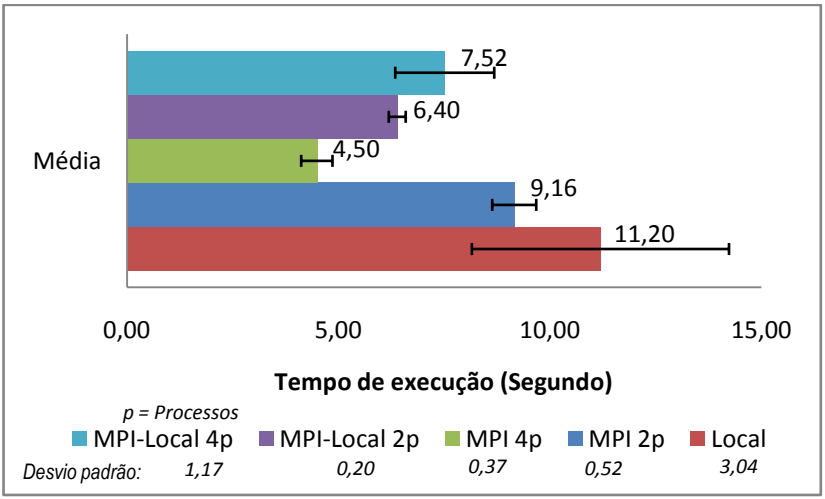

Figura 13. Média e desvio padrão dos tempos de execução dos programas de teste no segundo experimento.

Com o aumento do número de arquivos as configurações paralelas utilizadas no programa de testes apresentaram maior redução do tempo de execução em relação a configuração *Local*. O percentual de redução do primeiro experimento com a configuração *MPI 4p* foi de 51,16 %, e no segundo experimento foi de 59,86 %, uma diferença de 8,70 %.

Tabela 6. Cálculo do *Speedup* (aceleração) e eficiência.

Configurações	*Média*	*D. Padrão*	*Speedup*	*Eficiência*
MPI 2p	9,16	0,52	1,22x	0,61
MPI 4p	4,50	0,37	2,49x	0,62
MPI-Local 2p	6,40	0,20	1,75x	0,88
MPI-Local 4p	7,52	1,17	1,49x	0,37
Local = 11,20			*Sp = Ts/Tp*	*Ef = Sp/p*

A Tabela 6 apresenta melhores resultados neste experimento em relação a Tabela 5, principalmente para a configuração *MPI 4p*, com aumento de aceleração de 0,44 vezes e aumento de eficiência em 0,11 pontos. Mesmo obtendo melhores resultados no segundo experimento é necessário realizar testes em maior escala para determinar se de fato, existe uma tendência positiva quando um maior número de arquivos é submetido ao programa de teste.

5.2.3 Considerações sobre os Experimentos Realizados

A versão paralela do programa de teste apresentou os melhores resultados em duas configurações, *MPI-Local 2p* e *MPI 4p*, onde o emprego do número total de processadores físicos demonstra ser o melhor ajuste neste ambiente de teste. A configuração *MPI 2p* que utilizou os dois computadores disponíveis foi superada pela versão paralela executada localmente (*MPI-Local 2p*), o que revela não ser vantajoso distribuir a tarefa nesta situação.

Executar o processo de compilação de software de modo paralelo com o uso do Open MPI demonstrou ser interessante no ambiente de teste. Houve redução de mais da metade do tempo necessário para efetuar a tarefa na melhor configuração em comparação com a configuração que utilizou a versão não paralela do programa de testes.

6 CONSIDERAÇÕES FINAIS

Neste trabalho buscou-se uma alternativa para reduzir o tempo de compilação de programas de computador. Otimizar o uso dos recursos disponíveis em um ambiente computacional é uma proposta da computação paralela e distribuída, por isso este tipo de solução foi empregada neste problema.

Os resultados obtidos dentro do ambiente de teste foram satisfatórios e revelaram um potencial a ser explorado. O trabalho realizado com a plataforma Open MPI utilizou apenas os procedimentos básicos para criar um ambiente paralelo, porém há muito em relação a API que pode ser estudado.

O programa de teste demonstrou-se eficiente até mesmo ao utilizar os recursos de um único computador, ao dividir a tarefa no número de núcleos físicos disponíveis. Mesmo todos os testes com a versão paralela do programa de testes tendo superado a versão serial, algumas configurações não se apresentaram interessantes por representar pouca aceleração em vista dos recursos empregados, por isso escolher a melhor configuração conforme o ambiente disponível é fundamental para obter o melhor resultado.

A implementação Open MPI é utilizada em grandes clusters, como o RoadRunner da IBM, líder do top500 em 2008, uma lista que classifica os computadores mais velozes do mundo (PAULA; PRADO; CORNACCHIA, 2008). O RoadRunner alcançou em 2008 a marca histórica de 1 PFLOPS (*Peta FLoating-point Operations Per Second*), e foi desativado em 31/03/2013 para ser substituído por um computador menor, que custou menos da metade do preço do antecessor e ainda é um pouco mais rápido (IG, 2013). Mas além da aplicação do Open MPI em grandes clusters, a API também apresentou bons resultados mesmo em computadores IBM-PC, tal como o ambiente de teste utilizado neste trabalho. Isso representa uma área bastante interessante para ser estudada e aplicada em problemas de computação de alta performance.

6.1 PROPOSTAS PARA TRABALHOS FUTUROS

Nesta seção são listados algumas propostas para trabalhos futuros:

1. Comparar o Open MPI com outras implementações do padrão MPI;
2. Avaliar o Open MPI em ambiente de produção de software;
3. Testar a plataforma Open MPI em um ambiente computacional com mais componentes, submetendo aplicação a ser compilada com volume maior de arquivos.
4. Avaliar a infraestrutura de rede que conecta os computadores.

REFERÊNCIAS

AHO, A. V.; ULLMAN, J. D.; SETHI, R. **COMPILADORES: PRINCIPIOS, TECNICAS E FERRAMENTAS**. LTC, 1995. ISBN 9788521610571.

APPEL, A. W. **Modern compiler implementation in ML**. Cambridge university press, 1998. ISBN 0521582741.

BACELLAR, HILÁRIO VIANA. **Cluster: Computação de Alto Desempenho.** Universidade Estadual de Campinas, 2010.

BOINC. **BOINC.** Disponível em: <http://boinc.berkeley.edu/>. Acesso em: 20 jun. 2014.

CÁCERES, E. N.; MONGELLI, H.; SONG, S. W. **Algoritmos paralelos usando CGM/PVM/MPI: uma introduçao**. XXI Congresso da Sociedade Brasileira de Computaçao, Jornada de Atualizaçao de Informática, 2001. p.219-278.

CAMPAGNER, Carlos Alberto. **Média, desvio padrão e variância: Noções de estatística.** Disponível em: <http://educacao.uol.com.br/disciplinas/matematica/media-desvio-padrao-e-variancia-nocoes-de-estatistica.htm>. Acesso em: 23 jul. 2014.

COELHO, S. A. **Introdução a Computação Paralela com o Open MPI**. Simpósio Mineiro de Computação, p. 24-44, 2012.

COULOURIS, G., KINDBERG, T., DOLLIMORE, J. **Sistemas Distribuídos: Conceitos e Projeto. 4ª Edição.**: Editora Bookman 2007.

DANTAS, M. A. **Computação distribuída de alto desempenho: redes, clusters e grids computacionais**. Axcel Books, 2005. ISBN 8573232404.

PRICE, A. M. A.; TOSCANI, S. S. **Implementação de linguagens de programação: compiladores**. Sagra-Luzzatto, 2001. ISBN 9788524106392.

GABRIEL, Edgar et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Springer Berlin Heidelberg, 2004. p. 97-104.

GROPP, William et al. **A high-performance, portable implementation of the MPI message passing interface standard**. Parallel computing, v. 22, n. 6, p. 789-828, 1996.

HAZAN, Claudia; STAA, Av. Análise e Melhoria de um Processo de Estimativas de Tamanho de Projetos de Software. Monografias em Ciências da Computação, n. 04/05, 2004.

IG. **EUA desligam o supercomputador IBM Roadrunner.** 2013. Disponível em: <http://tecnologia.ig.com.br/2013-03-31/eua-vao-desligar-o-supercomputador-roadrunner.html>. Acesso em: 07 jul. 2014.

MOKARZEL, Fábio Carneiro; PANETTA, Jairo. **Reestruturação Automática de Programas Sequenciais para Processamento Paralelo**. II Simpósio Brasileiro de Arquitetura de Computadores-Processamento Paralelo (II-SBAC-PP)-Anais, v. 1, p. 1.1-7, 1988.

MORALES, D. F. D. G. **Compilação de código C/Mpi para C/Pthreads**. 2008.

NAVAUX, Philippe OA. **Introdução ao processamento paralelo**. RBC-Revista Brasileira de Computação, v. 5, n. 2, p. 31-43, 1989.

OPEN-MPI.ORG. **OpenMPI**. 2014. Disponível em: <http://www.open-mpi.org>. Acesso em: 24 mar. 2014.

PACHECO, Peter S. **Parallel programming with MPI**. Morgan Kaufmann, 1997.

PAULA, Andre C. de; PRADO, Gabriel L.; CORNACCHIA, Julio C. F.. **O supercomputador IBM RoadRunner.** 2008. Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2008/Trabalho/g18_texto.pdf>. Acesso em: 07 jul. 2014.

PEREIRA, Sandra Leandro; DE BRITO VIEIRA, Thiago Pereira. **Estudo da aplicação de um processo gerenciado de produção de software em MPEs**. Centro de Ciências Sociais Aplicadas. Universidade Federal da Paraíba. João Pessoa, 2006.

PUC-RIO. **A linguagem de programação LUA.** Disponível em: <http://www.lua.org>. Acesso em: 16/06/2014.

QUEIROZ, LUCAS P.; DE CARVALHO-JUNIOR, FRANCISCO H. **Avaliação do desempenho de operações coletivas em memória distribuída e compartilhada para implementações de MPI**. Universidade Federal do Ceará, 2007.

SNIR, M. et al. **MPI-The Complete Reference, Volume 1: The MPI Core**. MIT Press, 1998. 426 ISBN 0262692155.

SQUYRES, Jeffrey M.. **Open MPI.** Disponível em: <http://aosabook.org/en/openmpi.html>. Acesso em: 24 jun. 2014.

SQUYRES, Jeffrey M.; LUMSDAINE, Andrew. **The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms***. In: Component Models and Systems for Grid Applications. Springer US, 2005. p. 167-185.

STANDISH GROUP. **Chaos Manifesto: Think Big, Act Small**. The Standish Group International Inc, 2013.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems**. Prentice Hall Upper Saddle River, 2002.

APÊNDICE A – Código fonte do programa de teste paralelo utilizando a API Open MPI

```
#include "stdio.h"
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <mpi.h>

typedef struct listaArquivos
{
   int liContador;
   char** lsArquivos;
}listaArquivos;

struct listaArquivos get_lista_arquivos_c();

int main(int argc, char *argv[])
{
   int llRetorno, tid, nthreads, liIndice, liControle, liResto;
   int liInicio, liFim;
   char* lsComando, cpu_name;
   struct listaArquivos lstrListaArquivos;

   MPI_Init(&argc,&argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &tid);
   MPI_Comm_size(MPI_COMM_WORLD, &nthreads);

   lstrListaArquivos = get_lista_arquivos_c();

   liControle  = lstrListaArquivos.liContador / nthreads;
   liResto     = lstrListaArquivos.liContador % nthreads;

   if(tid == 0){
       printf("Total de arquivos: %i, Fator por processo: %i,
Resto: %i \n", lstrListaArquivos.liContador, liControle,
liResto);
   }

   cpu_name = (char *) calloc(80, sizeof(char));
   gethostname(cpu_name, 80);

   printf("Processo = %i na maquina = %s, de %i processes\n",
tid, cpu_name, nthreads);

   liFim       = (tid + 1) * liControle;
   liInicio    = liFim - liControle + 1;

   if((tid + 1 == nthreads) && liResto > 0) liFim += liResto;

   lsComando = (char *) calloc(80, sizeof(char));
```

```
   for(liIndice = liInicio; liIndice <= liFim; liIndice ++){
        printf("-- Processo %i, arquivo %s --", tid,
lstrListaArquivos.lsArquivos[liIndice]);
        strcpy(lsComando, "gcc -c ");
        strcat(lsComando,
lstrListaArquivos.lsArquivos[liIndice]);
        llRetorno = system(lsComando);
        if(llRetorno == 0){
                printf(" [OK] \n");
        }else{
                printf("Retorno da chamada: %i\n", llRetorno);
        }
   }

   MPI_Finalize();
   return(0);
}

struct listaArquivos get_lista_arquivos_c()
{
   int liIndice;
   char* lsExtencao;
   const char lsCaracter = '.';
   struct dirent **namelist;
   struct listaArquivos lstrListaArquivos;

   lstrListaArquivos.liContador = 0;
   liIndice = scandir(".", &namelist, 0, alphasort);

   if(liIndice < 0){
        perror("scandir");
   }else{
        lstrListaArquivos.lsArquivos = (char **) malloc(liIndice
* sizeof(char*));
        while(liIndice--){
                lsExtencao = strrchr(namelist[liIndice]->d_name,
lsCaracter);
                if(lsExtencao == NULL) lsExtencao = "/";
                if(strcmp(lsExtencao, ".c") == 0){
                        lstrListaArquivos.liContador ++;
                        lstrListaArquivos.lsArquivos[lstrListaArq
uivos.liContador] = (char *) malloc(strlen(namelist[liIndice]-
>d_name)+1);
                        strcpy(lstrListaArquivos.lsArquivos[lstrL
istaArquivos.liContador], namelist[liIndice]->d_name);
                        free(namelist[liIndice]);
                }
        }
        free(namelist);
   }
return lstrListaArquivos;
}
```